From (sequential) Haskell to (parallel) Eden: An Implementation Point of View

Silvia Breitinger, Ulrike Klusik and Rita Loogen*

Philipps-Universität Marburg, D-35032 Marburg, Germany
{breiting,klusik,loogen}@mathematik.uni-marburg.de

Abstract. The explicitly parallel programming language Eden adds a coordination level to the lazy functional language Haskell. This paper describes how a compiler and runtime system for Eden can incrementally be built on the basis of a compiler and runtime system for the computation language. The modifications needed in the compiler are restricted to specific orthogonal extensions. We show that Eden's design for distributed memory systems proves beneficial for the construction of a lean parallel runtime system.

1 Introduction

Due to the side effect freedom of the reduction semantics of functional languages it is possible to evaluate independent subexpressions in arbitrary order or in parallel. This *implicit parallelism* is semantically transparent and allows the automatic parallelization of functional programs. In available parallel functional systems like the Glasgow parallel Haskell (GpH) system [17] or the Nijmegen CLEAN system [19] the programmer is asked to place annotations in programs. These annotations show the compiler which subexpressions can potentially be evaluated in parallel. They only affect the runtime behaviour of programs.

Apart from the exploitation of implicit parallelism, explicit parallel constructs can be added in order to obtain more expressive parallel functional languages and to carry the benefits of functional languages into the world of parallel programming. Caliban [9] e.g. supports annotations for partitioning a program into a static net of processes which communicate via head-strict lazy streams. In *data-parallel functional languages* like Sisal [16], NESL [2] and SCL [7] special parallel structures and operations are provided to express data-parallel algorithms on a high-level of abstraction. *Concurrent functional languages* like Facile [8], Concurrent ML [14], Erlang [1] and Concurrent Haskell [12] add primitive constructs for the dynamic creation of concurrent processes (spawn or fork) and the exchange of messages between such processes (send and receive) to the functional kernel language. The main purpose of concurrent languages is however the definition of reactive systems and not the speeding up of computations by parallelism.

* The authors have been supported by the DAAD (Deutscher Akademischer Austauschdienst) in the context of a German-Spanish Acción Integrada.

The parallel functional language Eden [6,5] enables the programmer to explicitly express systems consisting of processes and interconnecting channels. This provides control over the granularity of processes and the communication topology. The placement of processes and the sending and receiving across communication channels will efficiently be handled by the system.

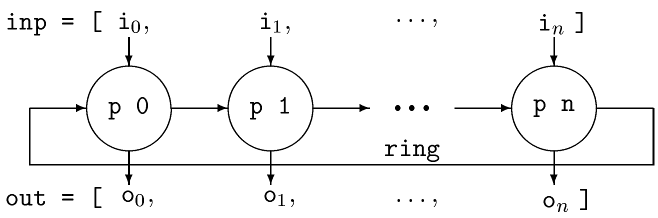
Eden is defined by adding a coordination language with explicit processes to the computation language Haskell [11]. It is tailored for distributed memory systems. The coordination language contains *process abstractions*, which define schemes for processes in a functional style, and *process instantiations*, which are expressions that generate processes and yield the respective outputs as their result. A process that maps inputs $in_1, \ldots, in_m$ to outputs $out_1, \ldots, out_n$ can be specified by the process abstraction

$$\texttt{process}\ (in_1, \ldots, in_m)\ \texttt{->}\ (out_1, \ldots, out_n)$$
$$\texttt{where}\ equation_1 \ldots equation_r$$

The optional `where` part of this expression is used to define auxiliary functions and common subexpressions which occur within the output expression.

Process abstractions have the type `Process a b` where `Process` is a newly introduced type constructor and the type variables `a` and `b` represent the input and output interface of the process, respectively. A process can have as input (respectively, output) tuples of channels and data structures of channels [4]. In the latter case, an annotation `<.>` is used to mark channels in types without changing the types themselves.

Example 1 (Process ring). Consider the following process structure which occurs in many parallel algorithms:



The whole system can easily be defined in Eden as a process that maps a list of input channels to a list of output channels and takes a process abstraction as a parameter that defines the behaviour of the component processes. The number of ring processes corresponds to the length of the input list:

```
procRing :: (Int -> Process (i,r) (o,r)) -> Process [<i>] [<o>]
procRing    p
  = process inp -> out
    where (out, ring) = createRing p 0 inp ring
```

Note that the type of the process abstraction makes clear that the component processes have two input and two output channels and that the integer identifier is not communicated but passed as a parameter before process creation. For the definition of the function `createRing` we need the concept of process creation.

Process instantiations create processes and their input and output channels. In such an expression, a process abstraction p is applied to a tuple of input expressions, yielding a tuple of outputs. The child process uses n independent threads of control in order to produce these outputs. Likewise, the parent introduces m additional threads that evaluate $input\_exp_1, \ldots, input\_exp_m$:

$$(out_1, \ldots, out_n) = p \ \# \ (input\_exp_1, \ldots, input\_exp_m)$$

Example 1 (cont'd): The function `createRing` can now be defined as follows:

```
createRing                  :: (Int -> Process (i,r) (o,r)) ->
                               Int -> [i] -> r -> ([o],r)
createRing p k []     ringIn =  ([], ringIn)
createRing p k (i:is) ringIn =  (o:os, ringOut)
   where (o, ringMid) = p k # (i,ringIn)  -- process instantiation
         (os,ringOut) = createRing p (k+1) is ringMid
```

The function `createRing` builds up the process ring element by element, as long as further inputs are available. Finally, the two ends of the ring connection are fused. The ring processes are created together with their communication interface by the process instantiations `p k # (i,ringIn)`.

Communication channels are unidirectional 1 : 1 connections which automatically transmit output expressions which have been evaluated to normal form before. Lists are transmitted in a *stream*-like fashion, i.e piecemeal. Upon process creation, communication channels for the transmission of inputs and outputs between parent and child will be established. An evaluation will be suspended if it depends on input that has not yet been received.

The evaluation of Eden processes is driven by the evaluation of the outport expressions, for which there is always demand. This rule overrides normal lazy evaluation in favour of parallelism. There will be a separate concurrent thread of execution for each output channel. Thus, we find in Eden two levels of concurrency: the concurrent or parallel evaluation of processes and the concurrent evaluation of different threads within processes. A process with all its outputs closed will terminate immediately. On termination its input channels will be eliminated and the corresponding outports in the sender processes will be closed.

Furthermore, each process immediately evaluates all *top level* process instantiations. This may lead to speculative parallelism and again overrules lazy evaluation to some extent, but speeds up the generation of new processes and the distribution of the computation.

No implicit or explicit sharing of information among processes is possible. All kinds of communication and synchronization have to be performed using communication channels. If a process abstraction contains global (free) variables on process creation, they will be replaced by their bindings. As the bindings may be unevaluated, this may lead to the duplication of work. Complex parameters should therefore be replaced by inputs.

The coordination constructs introduced up to now preserve referential transparency. This is not true for the two further constructs which support the definition of reactive systems in Eden but will not be handled in this paper: dynamic reply channels and predefined nondeterministic processes (see [5] for details).

In this paper we show how to extend the Glasgow Haskell compiler[1] (GHC), a publicly available fast implementation of Haskell, for Eden's parallel implementation. We describe the compilation of Eden and present a lean parallel runtime system which implements DREAM, the DistRibuted Eden Abstract Machine [3]. In the Eden parallel runtime system the process structures underlying parallel algorithms can exactly be modelled. Process abstractions determine the process interface and the environment in which the evaluation of the processes takes place. There is no need for a (virtual) shared memory or global address space. This simplifies the whole memory management including garbage collection. Several parts of the GUM (Graph reduction for a Unified Machine model) runtime system [18] for Glasgow parallel Haskell programs [17] have been reused for our parallel system. Our work is not restricted to Eden, a similar approach might be taken for other orthogonal extensions of sequential languages by a coordination language.

2 The Eden Compiler

The key idea of the GHC which forms the basis for the Eden compiler is "compilation by program transformation" [15]. Haskell programs are first transformed into Core Haskell, a minimal functional language in which all other Haskell constructs can be expressed. Core Haskell is mapped to a still functional abstract machine language STGL which has a simple denotational and operational semantics. The operational semantics is defined by the Spineless Tagless G-machine (STGM) which is an abstract description of the run-time system [13]. C is used as a portable target language of the compiler. Figure 1 shows the structure of GHC and the following extensions:

- The **parser** transforms the Eden constructs into Haskell expressions which contain functions defined in a Haskell module `EdenBase`.
- An additional compiler pass is introduced for the transformation of **STG to PEARL**. PEARL is an extension of the STG language by primitive parallel constructs. It enables Eden-specific optimizations and transformations.
- During **code generation** special code for the new PEARL expressions is generated.

2.1 Passing Eden programs through the GHC front end

From an implementation point of view, Eden's extensions to Haskell can be seen as "syntactic sugar" in the front end. We introduce for each parallel construct a *Haskell function* with the types chosen appropriately to ensure Eden's type restrictions.

We specify the Eden data type `Process a b` to distinguish between processes and functions and introduce Haskell functions `process` and `(#)`:

[1] see `http://www.dcs.gla.ac.uk/fp/software/ghc`

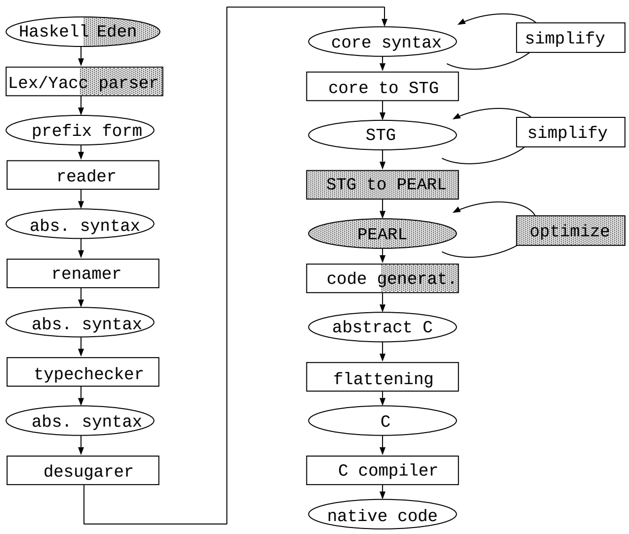


Fig. 1. Overview of GHC's compiler passes and our extensions

```
process :: (a -> b) -> Process a b
(#)     :: Process a b -> a -> b
```

In order to force the evaluation of output expressions to normal form and to model the sending of data via communication channels, the following classes are included in the module `EdenBase`:

```
class Eval a => NFData a where
  rnf   :: a -> ()
  rnf x =  x `seq` ()

class NFData a => Transmissible a where
  sendChan   :: a -> a
  sendChan x =  rnf x `seq` sendVal x `seq` x
```

The class `NFData` has been introduced in [17]. Its method `rnf` reduces expressions to normal form. The default implementation is the head normal form evaluation of the class `Eval`, which is sufficient for basic data types. For algebraic data types, the normal form is produced by forcing the normal form evaluation of all component expressions. The instance derivation tool described in [20] already provides rules for creating such instances.

The class `Transmissible` provides an overloaded function `sendChan` which is used to make explicit the sending of values across communication channels. In each of its instances, `sendChan` forces the evaluation of its argument and relies on primitive functions `sendVal`, `sendHead` and `closeStrm` to perform the communication[2]. The primitive functions are implemented by routines of the underlying message passing system. The default implementation of `sendChan` is used for all data types except lists. To communicate lists as streams the following instance declaration is used:

```
instance Transmissible a => Transmissible [a] where
  sendChan xs = sendStream xs `seq` xs
sendStream        :: Transmissible a => [a] -> ()
sendStream []     =  closeStrm
sendStream (x:xs) =  rnf x `seq` sendHead x `seq` sendStream xs
```

The parser recognizes a process abstraction of the form

```
process (i1, ..., im) -> (o1, ..., on)
where decls
```

and transforms it into the following Haskell expression using the definitions in the module `EdenBase`:

```
process ( \ (i1, ..., im) -> let decls
                             in  (sendChan o1, ..., sendChan on) )
```

A process instantiation of the form `pabs # (e1,...,em)` is transformed into `pabs # (sendChan e1,..., sendChan em)`.
The other Eden constructs are handled similarly.

To sum up, we have only modified the parser in order to embed Eden's additional expressions in Haskell syntax. The resulting Haskell program is passed through the subsequent, unchanged compiler passes, until STG syntax is reached. In particular, the original type inference algorithm is used to check the types of Eden programs.

2.2 Transforming STG into Pearl

Our intermediate language PEARL (Parallel Eden Abstract Reduction Language) is an extension of the STG language by parallel constructs. PEARL's operational semantics is given by DREAM (DistRibuted Eden Abstract Machine), an orthogonal extension of the STGM, formally specified in [3].

In PEARL, multithreading is expressed explicitly. The following syntax is used:

```
{expr || ... || expr}                    -- multithreading
{frees} \p {args} body                   -- process abstraction
{var, ..., var} = p # {expr || ... || expr}
                                         -- process instantiation
```

[2] The communication functions need no argument for the destination of the communication, because this is implicitly given in the corresponding outport definition.

The multithreading expression means that the component expressions should be evaluated concurrently, each by a separate newly created thread. It is used in process instantiations and in the body of process abstractions to implement the generation of threads. In the runtime system a primitive function `createThreads` is provided for that purpose. A process abstraction is represented by a lambda abstraction with free variables `frees`, flag `p` and parameters `args`. The process instantiations occur as new `letrec`-bindings in PEARL.

In fact, the definition of primitive functions for `process` and `(#)` would have been sufficient to generate process trees, in which a process only communicates with its parent or children. To implement arbitrary process topologies and to support the simultaneous instantiation of several connected processes, a special code generation is necessary. Such opportunities for optimizations are difficult to detect in Haskell or STG programs. This is the reason for introducing PEARL and the additional compiler pass. It allows to re-collect the information about processes and their interconnections.

The following translation schemes introduce the PEARL expressions for process abstraction and instantiation.

```
STG syntax:   var = process lf
              lf  = { frees } \n { args } body_exp
              lf :: t -> (t1, ..., tn)
-->
PEARL syntax: var    = { scheme, frees } \p { args }
                         let body' = { frees, args } \n { } body_exp
                         in  scheme body'
              scheme = {} \n x -> case x of
                                     (x1,..., xn) -> {x1 ||...|| xn}
```

The original process abstraction is transformed into a lambda abstraction with flag `p`. In order to introduce the multithreading expression in the body of the process abstraction, we extract the number of output channels from the type of the original lambda form[3] and apply a general scheme for the process body to the actual body expression.

```
STG syntax:   var = { p, arg } \n { } (#) p arg
              arg = { vi_1, ..., vi_n } \n { } (vi_1, ..., vi_n)
              p :: Process (it_1, ..., it_n) (ot_1, ..., ot_m)
-->
PEARL syntax: var = { vo_1,...,vo_m } \n { } (vo_1,...,vo_m)
              {vo_1,...,vo_m} = p # {vi_1 ||...|| vi_n}
```

In STGL, the function `(#)` indicates a process instantiation. The input tuple of this instantiation will be defined separately, as functions are applied to variables only. The type of the process abstraction is used to determine the number of

[3] In the GHC, the type information is still accessible in the STG program.

outputs of the instantiation. PEARL's process instantiation provides variables for the inputs and outputs. The multithreaded evaluation is made explicit. Of course, the output tuple of the process instantiation has to be reassigned to the original variable `var` as a tuple.

Naturally, the most important part of the back end is the translation into C. The compilation of STGL expressions and declarations remain unchanged. The PEARL transformation is determined by the parallel run-time system which is discussed in the following.

3 Eden's Parallel Runtime System

In this section we give an outline of the parallel runtime system of Eden which is based on the abstract machine DREAM [3]. We will show the advantages of the design decisions underlying Eden with respect to efficiency and ease of implementation in a distributed setting.

In **DREAM**, an Eden process consists of one or more concurrent threads of control. These evaluate different output expressions which are independent of each other and use a common heap that contains shared information (cf. Fig. 2).

Likewise, the state of a *process* includes information common to all threads and the states of the threads. The shared part includes the heap and an inport table, because input is shared among all threads. The state of a *thread* comprises the state of a sequential abstract machine and a specification of the associated outport referencing the connected inport. The interface of a process is visible in the following way: the outports are associated to threads (runnable or blocked) and the inports are enumerated in the inport table.

Communication channels are only represented by inports and outports which are connected to each other. An outport is just the global reference to an inport. These are the only global references needed. There are no explicit channel buffers, but the data is transferred from the producer's heap to the consumer's heap using the inport to find the location where the data should be stored.

We have already shown how we can benefit from our distinction of communication and coordination language by reusing a "state of the art" compiler for the computation language Haskell. In addition, we can take advantage of the runtime system GUM [18] for an implicitly parallel version of Haskell.

In the **parallel functional runtime system GUM**, the units of computation are called threads. Each *thread* evaluates an expression to weak head normal form. A thread is represented by a heap-allocated Thread State Object (TSO) containing slots for the thread's registers and a pointer to heap-allocated Stack Objects (SO). Each processor element (PE) has a pool of runnable threads.

The memory management in GUM is especially involved as it has to manage a virtual shared memory in which the shared program/data graph resides. The virtual shared memory requires global addresses which are (PE, local identifier) pairs. Local identifiers are mapped to local heap addresses using a Global Indirection Table (GIT).

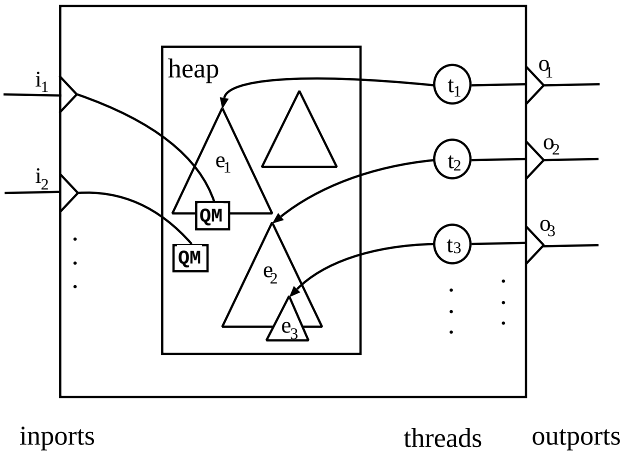


Fig. 2. A DREAM process

In order to transfer a subgraph from one PE to another, GUM uses sophisticated packing/unpacking algorithms, which guarantee that all the links back to the original graph are maintained and that the duplication of work is avoided. GUM uses a passive work distribution scheme, where PEs looking for work send out requests for work.

In principle, the GUM system is more complex than what is needed for Eden, as it supports a virtual shared memory and is demand driven. But the overall organization of the system, the interleaved evaluation of independent threads on a single processor, and the graph packing and unpacking algorithms can be incorporated in the Eden system, although they are used in different contexts.

The following aspects are fundamental in the design of a parallel runtime system:

1. communication between processes
2. multithreading inside of processes and scheduling
3. management of parallel activities
4. garbage collection

3.1 Communication

In contrast to most other parallel systems based on lazy functional languages, message data is evaluated and transmitted eagerly in Eden. This means that a one-message protocol is used for communication, i.e. one that dispenses with request messages for remote data. This feature is very important for the efficient implementation on distributed memory systems with considerable communication latencies. It furthermore alleviates the overlapping of computation and communication. The threads will be suspended less frequently than in a two message protocol, where a thread *must* be blocked after requesting the remote data. Nevertheless, message passing will not be done blindly, but mechanisms for controlling the flow of messages will be provided.

The sending of data across communication channels is initiated in special routines which correspond to the predefined communication functions in PEARL. The receipt of any messages is handled by a communication unit which puts the message contents into the heap and updates the inport table.

Graph Packing/Unpacking. To pack the result of an evaluation into a packet that can be sent across the network, GUM's graph packing and unpacking algorithms can be used. In Eden, it is ensured that a subgraph to be output contains only normal form data. This allows a substantial simplification of GUM's packing algorithm which checks for each closure node whether it contains normal form data or not in order to avoid the duplication of work. The effort for the globalization of shared non-evaluated closures (thunks) is saved. The GUM algorithms however have to be extended by the possibility to use more than one packet for the transmission of large subheaps.

3.2 Multithreading

In order to prevent multiple threads of the same process from evaluating the same shared expression, a synchronization mechanism is used, which can be found in most of the parallel runtime systems for functional languages. The thread that is the first to access a thunk, overwrites it with a so-called *black hole* or `queueMe`-closure. Subsequent threads attempting to access it will be blocked in a special queue and released when the first thread overwrites the `queueMe`-closure with the result of the computation.

GUM's thread management is perfectly appropriate for Eden's threads. The sequential evaluation of threads and the synchronization of threads within the same process need not be changed. The heap-allocated TSOs are extended by the outport specification, i.e. the global reference to the inport to which the result of the thread's computation must be sent.

Scheduling: Eden needs a fair scheduler for the evaluation of concurrent threads. Every logical PE runs a scheduler, which has the following tasks:

- perform local garbage collection (see Section 3.4)
- run the communication unit, which processes incoming messages
- run the active threads according to the scheduling strategy

The scheduling strategy is selected in a way such that it balances the speed of execution between senders and consumers. For efficiency reasons, it is vital to adapt the amount of output produced to the demand for information. In particular, we throttle senders which threaten to flood the heap of slower receivers.

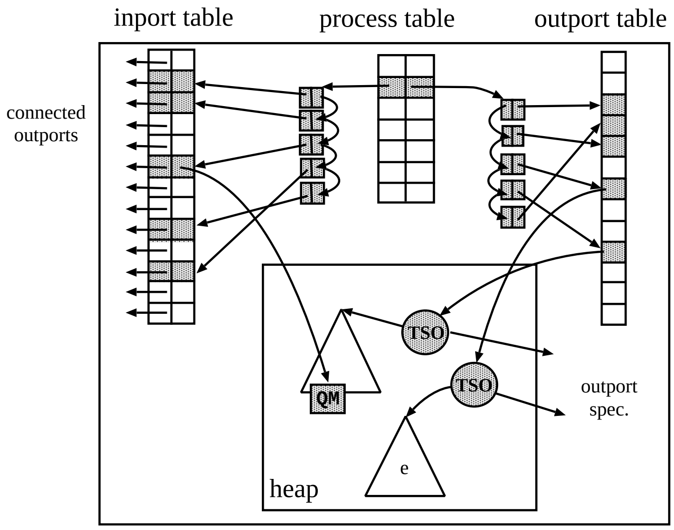


Fig. 3. Runtime tables

Assigning Eden processes to PEs. In order to map arbitrary process systems on a finite machine, we have to use interleaving of multiple processes on the same PE. In order to keep process creation as cheap as possible, we provide only one instance of the abstract machine per PE, which can execute several Eden processes concurrently. These processes share the heap, the scheduler and the runtime tables.

3.3 Management of parallel activities

Runtime tables. In order to establish the communication structure of the Eden process systems, new runtime structures have to be introduced. In every PE, we use the following tables to represent the communication channels and process interface information (see Fig. 3):

- The *inport table* maps locally unique identifiers of inports to heap addresses and the global references to the connected outports. Until the corresponding input data becomes available, a `queueMe`-closure is stored at the heap address. The outport references are used for the propagation of termination information.
- The *outport table* maps locally unique identifiers of outports to the corresponding thread (TSO) address. The outport table is used for system management, i.e. garbage collection, termination, error detection, etc.
- The *process table* contains for each process identifier the linked lists of inport and outport identifiers. Several Eden processes share the same abstract machine, and there is one entry in the process table for every process.

A *global reference* consists of the identifier of a remote PE and an identifier which is unique within this PE and will be used to index a runtime table.

Process creation. The evaluation of a process instantiation leads to the creation of a child process which will be spawned on a processor element selected by the load balancing mechanism. The local result of the process creation is the tuple of new inports, which will be filled by the child process. The code generated for process abstractions and process instantiations causes the following sequence of actions (see Fig. 4):

I The **parent** PE evaluates a thread that encounters a process instantiation `pabs # (in1, ..., inm)`. The process abstraction `pabs` and the tuple of input expressions `in1`, ..., `inm` will be available as heap closures, the addresses of which are given on the stack within the TSO of this thread.

II The **parent** PE

- generates `queueMe`-closures for the new *inports*, to which the outports of the new process will point, provides new identifiers for them and inserts them into the inport and process tables.
- provides new identifiers for the new *outports*, which have to be connected to the inports of the new process, inserts them into the outport and process tables correspondingly.
- composes a **createProcess** *message*, which contains the processor identifier, the whole subheap of the process abstraction[4] and the identifiers for the inports and outports of the process. On process creation the *complete* subheap representing the process abstraction is duplicated to build the heap of the new abstract machine of the new process. This ensures that the data dependencies between processes are restricted to the ones represented in the process interfaces.
  The global references to the inports and outports are pairs with the processor identifier and the respective port identifiers. The subheap can be packed using our simplified packing algorithm, because thunks can be copied.
- sends this message to the "child PE" determined by the process distribution and load balancing algorithm.
- creates new threads $t_1, \ldots, t_m$ for the evaluation of the input expressions for the new process, which are suspended until the new process has returned an acknowledgement message with the global references to the corresponding inports.
- returns the tuple with the *heap addresses* of the `queueMe`-closures which represents the (not yet available) result of the subprocess evaluation.

On receipt of the process creation message, the **child** PE

[4] If the subheap contains a `queueMe`-closure which represents a *shared expression* currently under evaluation by another thread, we copy the original closure nevertheless. This is possible because we use the so-called *revertable* black holes already implemented for GUM. If it however depends on an *inport*, blocking cannot be avoided.

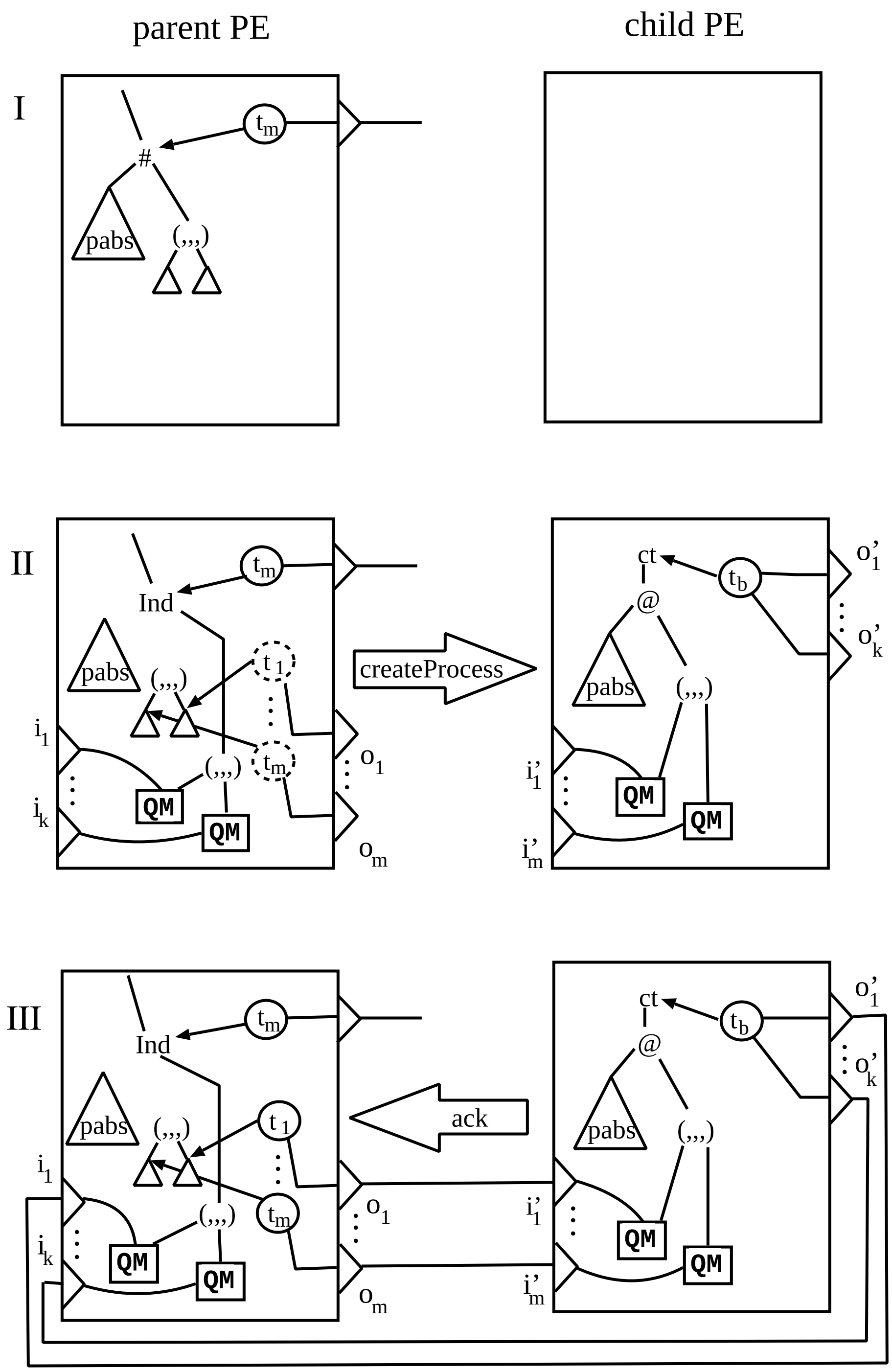


Fig. 4. Process instantiation

– generates `queueMe`-closures for the *inports* of the new process and adapts the inport and process table. The latter is extended by an entry for the new process. Note that the child PE can immediately store the connected outports of the parent PE in its inport table.
– generates new identifiers for the *outports* and extends the outport and process table accordingly.
– unpacks the *process abstraction* and starts the initial thread (t_b) of the process evaluation which will build up the process environment and then spawn the threads for the evaluation of the outport expressions via the primitive function `createThreads` (ct). In the meantime the outport specifications of these threads are stored in the state of the initial thread.

III The **child** PE sends an `ack` *message* to the parent process which contains a mapping between the global addresses of connected inports and outports. On receipt of the `ack` message, the **parent** PE writes the global references to the child's inports into the thread state objects of the outports thereby reactivating the suspended threads. Additionally, it stores the connection of the inports in its inport table.

Process distribution and load balancing. Eden requires an active process distribution algorithm. In the first version of the parallel runtime system we use a random distribution algorithm, i.e. a processor that instantiates a new process sends the `createProcess` message to a randomly selected PE. We plan to investigate more elaborate versions of distribution and balancing algorithms, which use bookkeeping about processor loads and information about process topologies and which choose a suitable substitute in case the selected PE is unable to accept the `createProcess` message.

3.4 Garbage collection and termination

There is no need for (general) global garbage collection of heap closures in the Eden runtime system, because there is no global address space. The local garbage collection proceeds as in the sequential system, but with additional roots in the evacuation phase. This extension is similar to the treatment in the GUM system where the entries of the global indirection table (GIT) are used as additional roots for local garbage collection.

An obvious thing to do would be to use the inports as additional roots in the Eden system, because these represent the only global references to the local heap. But this approach hinders the detection of input that is no longer consumed by any thread. For this reason, we decided to use the outports, or to be precise the TSO addresses within the outport table, as additional roots for local garbage collection. This corresponds to our view that there is always demand for output and it facilitates to detect that an inport (i.e. its heap location) is not referenced any more by any thread. In this case we can terminate the sending thread and sometimes even break dependency cycles between processes (see below).

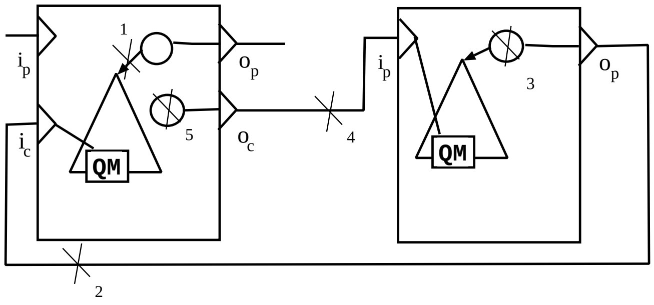


Fig. 5. Garbage collection of inports

Process termination. Processes without any outports will be terminated as they cannot contribute to the overall result of the program. This situation is detected, when an outport is closed, by checking whether the outport list of the corresponding process in the process table is empty. The inports of the process will be closed and termination messages will be sent to the connected outports.

There are two situations in which *an outport* is closed: if its thread terminates or if a termination message from the receiver process arrives. In the latter case the PE closes the addressed outport and kills the corresponding thread. This may lead to the termination of further processes. Likewise, there are two situations in which *an inport* is closed: if its process terminates or if it is not referenced any more. The latter is detected by the local garbage collection which checks whether the `queueMe`-closures of the inports have been reached from the roots.

The closing of inports whose `queueMe`-closures can no longer be accessed is an important vehicle for the termination of subsystems of processes which are no longer needed, but maintain mutual input/output connections. Many of such systems will shut down by the closing of inports, when the outports which connect these systems with the main system will be closed. Fig. 5 visualizes a scenario where a thread does no longer consume input data (1). The inport will be closed after garbage collection and a termination message will be sent to the corresponding outport (2). This leads to the termination of the associated thread (3) and so on.

4 Related Work

In parallel functional runtime systems like Haskell's GUM [18] and CLEAN's PABC (Parallel ABC machine) [10] the task of a process is the evaluation of an expression to weak head normal form. The whole computation is demand driven. The systems provide a global address space, sometimes called virtual shared memory. Whenever the evaluation needs the contents of a global address, a request message is sent to the processor element holding that address. In the PABC machine such request messages trigger the evaluation of the expression at the global address. In GUM the graph at the global address is packed and transferred to the requesting processor element. Also process distribution is usually done on demand, i.e. processor elements with empty work pools send requests for work to other processor elements.

In the specification of a parallel process graph in a functional language with parallelism annotations the communication channels will be modelled by function parameters, in implementation terms global addresses, and the request/answer mechanism, a two message protocol, will be adopted for "communicating values along the communication lines" of a process graph. It depends on the runtime system where expressions will be evaluated and when communication takes place. The Caliban system [9] tries to overcome this loss of topology information by extracting process net information at compile time and translating it into a call to a special system primitive called `procnet` which implements the run-time parallelism. This is only possible because the Caliban system is restricted to static process systems.

Data-parallel languages like Sisal [16], NESL [2] and SCL [7] provide only predefined parallel operations and skeletons, which can be implemented by adding parallel primitives to a sequential implementation. The development of a specific parallel abstract machine is not required. Concurrent functional languages like Facile [8], Concurrent ML [14], Erlang [1] and Concurrent Haskell [12] are based on low level parallel primitives, which are implemented on a shared memory base using fair schedulers to switch between the concurrent activities. Distributed implementations are feasible, but require the implementation of a virtual shared memory as in the runtime systems of functional languages with implicit parallelism. The low level process model of these languages obstructs the abstraction of a communication structure which could easily be mapped on a distributed system.

5 Conclusions

As Eden extends Haskell by a coordination language which allows the explicit definition of parallel process systems, an obvious approach to the implementation of Eden is to extend a Haskell compiler and runtime system. In this paper we have shown that the front end of the Glasgow Haskell compiler can be reused with minor extensions for the compilation of Eden into STGL. The back end of the compiler and the runtime system require changes, which are however designed as orthogonal additions to the existing implementation. Kernel parts of the parallel functional runtime system GUM could be reused and even simplified. The Eden parallel runtime system has recently been implemented using MPI. A detailed elaboration of this implementation, together with specific optimizations, is left for future work.

Acknowledgements. We thank Kevin Hammond, Hans-Wolfgang Loidl, Phil Trinder, Yolanda Ortega, Ricardo Peña, Steffen Priebe, and Mari Plümacher for valuable discussions.

References

1. J. Armstrong, M. Williams, and S. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1996.
2. G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.
3. S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM - the DistRibuted Eden Abstract Machine. In C. Clack, T. Davie, and K. Hammond, editors, *Symposium on the Implementation of Funct. Lang. 1997, St. Andrews, selected papers*, LNCS 1467. Springer, 1998.
4. S. Breitinger and R. Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Functional Programming*, 1997.
5. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden — Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996.
6. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS).* IEEE Press, 1997.
7. J. Darlington, Y.-K. Guo, H. To, and J. Yang. Functional skeletons for parallel coordination. In *EURO-PAR '95 Parallel Processing*, LNCS 966. Springer, 1995.
8. A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *Journal of Parallel Programming*, 18(2), 1989.
9. P. Kelly. *Functional Programming for Loosely Coupled Multiprocessors.* Pitman, 1989.
10. M. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, Katholieke Universiteit Nijmegen, 1996.
11. J. Peterson and K. Hammond (eds.). Report on the programming language Haskell: a non-strict, purely functional language, version 1.4. Technical Report YALEU/DCS/RR-1106, Yale University, 1997.
12. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Symp. on Principles of Programming Languages (POPL)*, 1996.
13. S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2), 1992.
14. J. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1991.
15. A. Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Glasgow University, Department of Computing Science, 1995.
16. S. K. Skedzielewski. Sisal. In B. Szymanski, editor, *Parallel Functional Languages and Compilers.* ACM Press, 1991.
17. P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.
18. P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.
19. M. van Eekelen and R. Plasmeijer. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.
20. N. Winstanley. Reflections on Instance Derivation. In *Glasgow Workshop on Functional Programming*, 1997.